Linux Kernel Training

Concurrency and Synchronization

Kernel Internals

Agenda

1. Kernel threads
   - Manual creation and management
1. Atomic operations
2. Per-CPU data
3. Spinlocks
4. Semaphores, mutexes, rt-mutexes
5. Task completion synchronization
6. Read-write locks, sequential locks

Kernel threads management

- include/linux/kthread.h
    - **kthread_create**(threadfn, data, namefmt, arg...) - <macro> create new thread;
    - **kthread_run**(threadfn, data, namefmt, ...) - create and wake a thread;
    - Completion synchronization:
        - int **kthread_stop**(struct task_struct *k) - wakes a thread, notify it to stop and wait for it;
        - bool **kthread_should_stop**(void) - verify if the current thread was requested to stop;
        - int **kthread_park**(struct task_struct *k)
        - bool **kthread_should_park**(void)
        - void **kthread_parkme**(void)

All kernel threads are created as processes forked from **kthreadd** *(see `**ps -ef**`)*

Threads under the hood

- include/linux/sched.h
  - struct **task_struct** - description of (kernel thread, userspace process or thread);
  - Task_struct <-> PID conversion:
    - pid_t **task_pid_nr**(struct task_struct *tsk)
    - struct task_struct ***find_task_by_vpid**(pid_t nr)
  - Task configuration:
    - void **set_user_nice**(struct task_struct *p, long nice)
    - int **sched_setscheduler** (struct task_struct *p, int policy, const struct sched_param *param) - set the scheduling policy;
  - Wakup:
    - void **wake_up_new_task**(struct task_struct *tsk)
    - int **wake_up_process**(struct task_struct *tsk)

Threads - continuation

- include/linux/sched/task.h
  - pid_t **kernel_thread**(int (*fn)(void *), void *arg, unsigned long flags) - API for creation of kernel threads based on **_do_fork**() call - the same way as for userspace processes;

- arch/${ARCH}/include/asm/current.h
  - **current** - <macro definition> pointer to task_struct of currently running process (per CPU core);

Preemption control

- include/linux/irqflags.h
    - **local_irq_disable**()
    - **local_irq_enable**()
    - **local_irq_save**(flags)
- include/linux/preempt.h
    - **in_interrupt**()
    - **in_atomic**()
    - **preempt_disable**()
    - **preempt_enable**()

Also see Documentation/memory-barriers.txt

Atomics

Atomic operations on **atomic_t** and **atomic64_t** usually defined in

- arch/${ARCH}/include/asm/atomic.h
  - **ATOMIC_INIT**(i)
  - int **atomic_read**(const atomic_t *v)
  - void **atomic_set**(atomic_t *v, int i)
  - void **atomic_add**(int i, atomic_t *v)
  - int **atomic_xchg**(atomic_t *v, int new)
  - etc.

See Documentation/core-api/atomic_ops.rst for development rationale.

- arch/${ARCH}/include/asm/bitops.h
  - void **set_bit**(long nr, unsigned long *addr)
  - bool **test_and_change_bit**(long nr, unsigned long *addr)
  - int **ffs**(int x) - find first set
  - etc.

Per-CPU variables

Optimization of cache utilization

- include/linux/percpu.h
  - arch/${ARCH}/include/asm/percpu.h
  - include/linux/percpu-defs.h
  - **DEFINE_PER_CPU**(type, name), **alloc_percpu**(type)
  - **get_cpu_var**(var), **get_cpu_ptr**(var);
  - **put_cpu_var**(var), **put_cpu_ptr**(var);
  - **per_cpu**(var, cpu), **per_cpu_ptr**(ptr, cpu) - to access variable for other CPU;
  - **this_cpu_**operations - atomic operations directly in allocated per-cpu area;
  - **this_cpu_ptr**(ptr) - returns pointer, which allows direct operations;

See Documentation/this_cpu_ops.txt

Spinlock

The most basic locking primitive. Blocked thread (which tried to take already acquired lock) executes a busy wait loop until the lock is released.

- include/linux/spinlock.h
  - **spinlock_t**
  - **DEFINE_SPINLOCK**(x)
  - void **spin_lock**(spinlock_t *lock)
  - int **spin_trylock**(spinlock_t *lock)
  - void **spin_unlock**(spinlock_t *lock)
  - **spin_lock_irqsave**(lock, flags)
  - void **spin_unlock_irqrestore**(spinlock_t *lock, unsigned long flags)

- Spinlocks should not be held for a long time.
- Blocking operations may not be used when holding a spinlock.
- Spinlocks are not recursive.

See Documentation/locking/spinlocks.txt

Semaphore

Synchronization primitive for long time locks with context switch.

- include/linux/semaphore.h
    - struct **semaphore**
    - void **sema_init**(struct semaphore *sem, int val) - dynamically initialize counting semaphore;
    - **DEFINE_SEMAPHORE**(name) - define and statically initialize binary semaphore;
    - void **down**(struct semaphore *sem) - take one;
        - also available interruptible, trylock, timeout variants;
    - void **up**(struct semaphore *sem) - release;

Semaphore is a typically controls access to limited resources.

Mutex

Blocking mutual exclusion lock.

- include/linux/mutex.h
  - struct **mutex**
  - **mutex_init**(mutex) - dynamically initialize mutex;
  - **DEFINE_MUTEX**(mutexname) - define and statically initialize mutex;
  - void **mutex_lock**(struct mutex *lock)
    - also available nested (lock order validation) and interruptible variants;
  - int **mutex_trylock**(struct mutex *lock)
  - void **mutex_unlock**(struct mutex *lock)

- Only one task can hold the mutex at a time.
- Only the owner can unlock the mutex.
- Recursive locking/unlocking is not permitted.
- Mutexes may not be used in interrupt contexts.

See Documentation/locking/mutex-design.txt

RT-mutex

Real time mutexes extend the semantics of simple mutexes by the priority inheritance protocol to avoid unlimited priority inversion.

- include/linux/rtmutex.h
  - struct **rt_mutex**
  - **DEFINE_RT_MUTEX**(mutexname)
  - void **rt_mutex_lock**()
  - int **rt_mutex_trylock**()
  - void **rt_mutex_unlock**()

See Documentation/locking/rt-mutex.txt and Documentation/locking/rt-mutex-design.txt

Completions

Generic wait-notify synchronization

- include/linux/completion.h
  - struct completion;
  - DECLARE_COMPLETION(work) - declare and initialize a completion structure;
  - void wait_for_completion(struct completion *x) - locks on specified task;
    - interruptible and timeouts variants are also available
  - void complete(struct completion *x) - wake up a single waiting thread;
    - void complete_all(struct completion *x)

Waiting for completion is a typically sync point, but not an exclusion point.

See Documentation/scheduler/completion.txt

Read-write locks

Locks with privileged access for read-only operations.

- include/linux/rwlock.h - Now deprecated in favor to RCU-locks
  - **read_lock**(), **read_unlock**()
  - **write_lock**(), **write_unlock**()
  - other API variants as for spinlock are also available.
- include/linux/rwsem.h
  - struct **rw_semaphore**
  - **DECLARE_RWSEM**(name)
  - **down_read**(), **up_read**()
  - **down_write**(), **up_write**()
- include/linux/rcupdate.h
  - Read-Copy Update mechanism for mutual exclusion.
  - See Documentation/RCU/rcu.txt

Sequential lock

Lightweight and scalable lock for use with many readers and a few writers. Based on spinlock and exclusive access counter.

- include/linux/seqlock.h
  - **seqcount_t**
  - Initialization:
    - **DEFINE_SEQLOCK**(x), **seqlock_init**(x)
  - Reader critical section:
    - unsigned **read_seqbegin**(const seqlock_t *sl)
    - unsigned **read_seqretry**(const seqlock_t *sl, unsigned start)
  - Writer critical section:
    - **write_seqlock**(), **write_sequnlock**()
    - **write_seqlock_irq**(), **write_sequnlock_irq**()
  - Reader exclusive critical section:
    - **read_seqlock_excl**(), **read_sequnlock_excl**()

Reader section example:

```
DEFINE_SEQLOCK(lock);
unsigned seq;
do {
    seq = read_seqbegin( &lock );
    /* work here */
} while read_seqretry( &lock, seq );
```

Be wise